📋 API CA Prep Day 2: JWT Authentication & User Integration

Table of Contents

1. Core Requirements

Today you will build authentication into your Event Planner API. This involves:

- Adding a `User` model with authentication details.
- Refactoring existing models and services to include users.
- Creating JWT-based authentication routes (`/auth`).
- Writing integration tests with Supertest.

Required packages:

```
npm install jsonwebtoken jsend
npm install --save-dev supertest
```

2. Models Refactoring

User Model

Create a new `User` model with the following fields:

Field	Type	Constraints
id	INT	Primary Key, Auto Increment
email	STRING	Required, Valid email format
encryptedPassword	BLOB	Required
salt	BLOB	Required

Relations:

- `User` has many `Events`

- `Event` belongs to `User` (FK: `userId`)

Event Model Update

- Add `userId` field to the `Event` model as a foreign key referencing `User`.

3. Password Helper

Use the provided `passwordHelper.js` to handle password encryption and validation. (Code provided separately)

4. Test Database Initialization

Create a helper function `testDbInit(db)` (sample code provided separately) that:

- Syncs the DB
- Seeds EventTypes
- Seeds two Users (with hashed passwords)
- Seeds Events linked to seeded users

Refactor your existing test setup to use this helper.

NB: Make sure all your event service tests still pass.

5. Service Layer Refactoring

Update your `EventService`:

- All test cases (valid and invalid) now include `userId` (use `userId: 1` for simplicity).

Example valid test data:

```
const validData = {
  title: 'Test Event 3',
  date: '2025-06-11',
  location: 'Oslo',
  eventTypeId: 1,
  userId: 1
}
```

New Service Method

Add the method `getEventsForUser(userId)` to your `EventService`:

```
async getEventsForUser(userId) {
  // Implement
}
```

Write a test case verifying it returns correct events for a specific user.

6. JWT Authentication

Create a new route file: `/routes/auth.js`. Implement the following endpoints:

- **POST /auth/signup**:
  - Body: `{ email, password }`
  - Validate input (email format, fields required)
  - Creates a user, stores encrypted password
  - Returns: `201 Created` + new user object (no password fields)
- **POST /auth/login**:
  - Body: `{ email, password }`
  - Validate input
  - Verifies password
  - Returns JWT token (`sub`: userId, expiry: 15 mins)
- **GET /auth/protected**:
  - Protected endpoint (requires valid JWT in `Authorization: Bearer <token>` header)
  - Returns user details or simple protected resource message
  - Implement the token extractin and validation as middleware (this repo from last week can aid you)

JWT Details:

- Secret key stored in `.env` file.
- Payload structure: `{ sub: userId }`

Response Formatting with JSend

Use the `jsend` package to format all your authentication endpoint responses consistently:

- On successful operations (signup, login, protected route), use:

```
res.status(200).json(jsend.success({ /* your data */ }))
```

- For client-side validation errors or failures (missing fields, invalid credentials), use:

```
res.status(400/404).json(jsend.fail({ /* error details */ }))
```

- For authentication errors (invalid or missing JWT), use:

```
res.status(401).json(jsend.fail({ /* error details */ }))
```

Ensure all auth responses adhere to this JSend structure for clarity and consistency.

7. Integration Testing

Write tests using Supertest for all auth endpoints:

✅ Good Paths

- **Signup:** Valid credentials (returns 201 and user data)
- **Login:** Correct credentials (returns JWT)
- **Protected:** Valid token allows access

❌ Bad Paths

- **Signup:**
  - Invalid email format: returns `400`
  - Missing fields (email/password): returns `400`
- **Login:**
  - Incorrect credentials: returns `401`
  - Missing fields (email/password): returns `400`
  - Email does not exist: returns `404`
- **Protected:**
  - Missing token: returns `401`
  - Invalid token signature: returns `401`
  - Expired token: returns `401`

⚠️ NOTE:

In real-world scenarios, you would also create unit tests for the `UserService` methods. However, for this day, your integration tests adequately cover the functionality.

8. Sample Code

`passwordHelper.js`:

```
// ADAPTED FROM API LESSON 5.2 (Authentication section)

const crypto = require('crypto') // Built-in Node.js module for cryptography

/**
 * Hashes a plain text password securely using PBKDF2 algorithm.
 *
 * How it works:
 * 1. Generates a random salt (randomBytes).
 * 2. Uses PBKDF2 (Password-Based Key Derivation Function 2) to hash the password
```

```
+ salt.
 * 3. Returns the salt and hashed password as Buffers (binary format), ready to be
stored in the DB.
 *
 * Why use salt?
 * Salt ensures that even if two users have the same password, their hashed
results will be different.
 *
 * @param {string} plainPassword - The password entered by the user.
 * @returns {Promise<{ salt: Buffer, encryptedPassword: Buffer }>} - Salt and
hashed password.
 */
function hashPassword(plainPassword) {
  return new Promise((resolve, reject) => {
    // 16 random bytes = 128-bit salt, strong enough for most uses.
    const salt = crypto.randomBytes(16)

    // PBKDF2 Parameters:
    // - password
    // - salt
    // - iterations: 310,000 (higher = more secure but slower)
    // - key length: 32 bytes (256 bits)
    // - algorithm: sha256 (good modern choice)
    crypto.pbkdf2(plainPassword, salt, 310000, 32, 'sha256', (err, hashedPassword)
=> {
      if (err) return reject(err)

      // Return salt + hashed password (both as Buffers)
      resolve({
        salt,
        encryptedPassword: hashedPassword
      })
    })
  })
}

/**
 * Compares a plain text password with an encrypted password.
 *
 * How it works:
 * 1. Re-hashes the provided plain password with the same salt.
 * 2. Compares the result with the stored hashed password using a timing-safe
comparison.
 *
 * Why timingSafeEqual?
 * Prevents timing attacks that could reveal information based on how long the
comparison takes.
 *
 * @param {string} plainPassword - The password entered by the user.
 * @param {Buffer} encryptedPasswordBuffer - Stored hashed password (from
database).
 * @param {Buffer} saltBuffer - Stored salt (from database).
 * @returns {Promise<boolean>} - True if password matches, false otherwise.
 */
```

```
function comparePassword(plainPassword, encryptedPasswordBuffer, saltBuffer) {
  return new Promise((resolve, reject) => {
    crypto.pbkdf2(plainPassword, saltBuffer, 310000, 32, 'sha256', (err,
hashedPassword) => {
      if (err) return reject(err)

      // Securely compare hashed passwords
      resolve(crypto.timingSafeEqual(hashedPassword, encryptedPasswordBuffer))
    })
  })
}

module.exports = {
  hashPassword,
  comparePassword
}
```

testDbInit.js:

```
const { hashPassword } = require('./passwordHelper')

async function initTestDb(db) {
  await db.sequelize.sync({ force: true })

  // Seed EventTypes
  await db.EventType.bulkCreate([
    { id: 1, name: 'Conference' },
    { id: 2, name: 'Meetup' },
    { id: 3, name: 'Workshop' },
    { id: 4, name: 'Seminar' }
  ])

  // Seed Users with hashed passwords
  const user1Password = await hashPassword('Password123')
  const user2Password = await hashPassword('Secret456')

  await db.User.bulkCreate([
    {
      id: 1,
      email: 'user1@example.com',
      encryptedPassword: user1Password.encryptedPassword,
      salt: user1Password.salt
    },
    {
      id: 2,
      email: 'user2@example.com',
      encryptedPassword: user2Password.encryptedPassword,
      salt: user2Password.salt
    }
  ])

  // Seed Events linked to users
```

```
  await db.Event.bulkCreate([
    { id: 1, title: 'Test Event 1', date: '2025-05-01', location: 'Oslo',
eventTypeId: 1, userId: 1 },
    { id: 2, title: 'Test Event 2', date: '2025-06-10', location: 'Bergen',
eventTypeId: 2, userId: 2 }
  ])
}

module.exports = { initTestDb }
```